Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera de Ingeniería en Informática

Extensión de funcionalidades de la herramienta PeabraiN

Iván Pamplona Cetina

Director: Ricardo J. Rodríguez Fernández

Ponente: Jorge Júlvez Bueno

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2014
Curso 2013/2014

A mi mujer Laura y a mi hijo Iker.

Stay hungry, stay foolish.
Steve Jobs

Agradecimientos

A mi mujer,
A mis padres,
A Ricardo por su paciencia y profesionalidad,
A Jorge Júlvez y José Merseguer,
A mis amigos Aitor y Juan Antonio,
A mis compañeros de promoción.

GRACIAS.

Extensión de funcionalidades de la herramienta PeabraiN

RESUMEN

PeabraiN es una herramienta para análisis de propiedades y simulación de Redes de Petri estocásticas que cumple perfectamente con su cometido para operaciones de programación lineal, pero se encuentra en su primera versión y tiene grandes posibilidades de mejora. Igualmente sucede con su simulador estocástico, que realiza simulaciones de manera correcta, pero existe gran margen de mejora en los tiempos que emplea para realizar dichas simulaciones, sacrificando ligeramente la exactitud de los resultados en mejoría del tiempo cálculo.

El proyecto que se expone en esta memoria está enfocado en dos partes diferenciadas. Por un lado se estudia y mejora la herramienta PeabraiN, a nivel de funcionalidades, y por otro lado se estudia y mejora el simulador estocástico de Redes de Petri que proporciona PeabraiN.

El resultado obtenido después analizar ambas partes han sido tres mejoras que aportan a PeabraiN una mayor personalización, interactividad y comunicación con el usuario además de una mejora en la simulación estocástica. Concretamente estas mejoras son: implementación de un selector de solucionadores de programación lineal que permita elegir el solucionador que se desee para realizar los cálculos, un CLI (*Command Line Interface*) que permita realizar los cálculos a través de comandos prescindiendo de la interfaz gráfica, y finalmente una aplicación Web que permita utilizar la herramienta a través de Internet. La cuarta mejora implementa un nuevo método de simulación estocástica que mejora considerablemente el tiempo de simulación.

Índice

1. Introducción **1**
1.1. Organización del documento . . . 1
1.2. Ámbito y motivación . . . 1
1.3. Objetivo . . . 2
1.4. Métodos y herramientas empleadas . . . 3
1.5. Fases del trabajo . . . 4

2. Conceptos previos **5**
2.1. UML . . . 5
2.1.1. Diagrama de clases . . . 5
2.1.2. Diagrama de secuencia . . . 6
2.2. Patrones de diseño . . . 7
2.3. Red de Petri . . . 9

3. Mejoras en la funcionalidad de PeabraiN **11**
3.1. Selector de solucionadores de Programación Lineal . . . 12
3.1.1. Análisis . . . 12
3.1.2. Diseño . . . 14
3.1.3. Configuración . . . 14
3.2. Línea de comandos (*Command Line Interface*, CLI) . . . 16
3.3. Cliente Web . . . 21
3.4. Extras . . . 23
3.5. Código fuente . . . 23

4. Mejoras en la computación de PeabraiN **25**
4.1. Método de simulación estocástica actual . . . 26
4.2. Nuevo método de simulación estocástica . . . 26
4.3. Comparativa de métodos . . . 28

5. Trabajo Relacionado **33**

6. Conclusiones **35**

A. Fases de Desarrollo **39**
A.1. Diagrama de *Gantt* 39

B. Código fuente del simulador estocástico aproximado (método Tau-Leaping) **41**

Índice de figuras

2.1. Ejemplo de Diagrama de clases UML. 6
2.2. Ejemplo de diagrama de secuencia UML. 7
2.3. Ejemplo de Red de Petri. 10

3.1. Arquitectura de software de PeabraiN (extraído de [RJM12]). 12
3.2. Integración de PeabraiN con Pipe2 (extraído de [RJM12]). 12
3.3. Opción de selección de solucionadores. 14
3.4. Diagrama de clases de selector de solucionadores. 15
3.5. Diagrama de clases del patrón *Singleton*. 15
3.6. Ejemplo de configuración del fichero ".solverselector.conf". 16
3.7. Diagrama de clases UML del CLI (*Command Line Interface*). 17
3.8. Diagrama de secuencia UML del CLI. 18
3.9. Ejemplo de ayuda para listar los módulos disponibles en PeabraiN a través del CLI. 19
3.10. Atributos de la clase *MyParamValue*. 20
3.11. Ejemplo de ayuda para el módulo *PerformanceEstimation*. 20
3.12. Diseño de la aplicación PeabrainWEB de una única ventana. 22
3.13. Imagen de la aplicación PeabraiN Web. Resultado final. 22
3.14. Categorización de la lista de módulos disponibles en Pipe2. 23

4.1. Red de Petri "cinema.xml". 29
4.2. Red de Petri "SDBS.xml". 29
4.3. Comparativa de tiempos de ejecución de ambas redes utilizando los dos métodos. 31
4.4. Comparativa de *throughput* (rendimiento) de ambas redes utilizando los dos métodos. 32
4.5. Error relativo de *throughput* para RdP "cinema.xml" y "SDBS.xml". . . . 32

A.1. Diagrama de Gantt del desarrollo del proyecto. 39
A.2. Distribución del esfuerzo en las diferentes fases del proyecto. 40

Índice de tablas

3.1. Solucionadores de Programación Lineal. 13

4.1. Tabla de algoritmos de simulación estocástica. 25
4.2. Resultados numéricos de la Red de Petri *cinema.xml*. 30
4.3. Resultados numéricos de la Red de Petri *SDBS.xml*. 30

A.1. Horas dedicadas. 39

Capítulo 1

Introducción

En este capítulo introductorio se van a tratar los aspectos más generales relacionados con el Proyecto, dentro de los cuales se encuentran los motivos que me llevaron a realizar este PFC, el ámbito en el que se enmarca, los conceptos que serán utilizados para su elaboración, los objetivos definidos y las fases en las que se ha organizado el trabajo.

1.1. Organización del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del Proyecto; y los apéndices, donde se amplía la información de ciertos puntos relevantes.

El Capítulo 1 expone los objetivos del proyecto e introduce el documento. El Capítulo 2 define alguno de los conceptos previos que servirán de ayuda para comprender el resto del documento, como UML, Patrones de diseño, Red de Petri. A continuación, el Capítulo 3 explica todas las mejoras llevadas a cabo a nivel funcional en PeabraiN. El Capítulo 4 analiza, desarrolla y compara un nuevo método de simulación estocástica aproximada con respecto al método de simulación exacta que viene por defecto con PeabraiN. El Capítulo 5 muestra los trabajos relacionados con este PFC, y por último, el Capítulo 6 detalla las conclusiones generales extraídas de la realización de este PFC.

Respecto a los Apéndices, el Apéndice A es donde se hace balance del esfuerzo temporal empleado en la realización del PFC, y el Apéndice B muestra el código Java del simulador estocástico aproximado.

1.2. Ámbito y motivación

Este proyecto fin de carrera (PFC) está relacionado con la herramienta PeabraiN [RJM12], que fue desarrollada por el Grupo de Ingeniería de Sistemas de Eventos Discretos (GISED) [GIS] de la Universidad de Zaragoza. PeabraiN está accesible en `http://webdiis.unizar.es/GISED/?q=tool/peabrain`, y es un software que permite realizar el cálculo de todas aquellas propiedades de Redes de Petri [Mur89] que se pueden

expresar mediante problemas de optimización [CS92, Cam98, RJM13], como por ejemplo estimaciones de rendimiento, optimización de recursos, cotas estructurales de marcado y habilitación y ratios de visita de transiciones. Esta herramienta también permite simular Redes de Petri estocásticas.

El PFC se presenta interesante y motivador porque está dividido en varias partes que obligan a conocer diferentes áreas de la informática para poder llevarlo a cabo, además de que amplía considerablemente las funcionalidades de la herramienta PeabraiN, limitada actualmente a realizar cálculos de programación lineal utilizando un solo solucionador de programación lineal (PL), a pesar de existir multitud de ellos, y a la interacción mediante interfaz gráfica con el usuario. Algunos de los campos que hay que conocer y que hacen atractivo el proyecto son las Redes de Petri, Programación Java, Programación Web dinámica (PHP y Ajax) y la Simulación Estocástica, entre otros. Esta diversidad de elementos englobados en el proyecto se debe a las diferentes fases en que se subdivide.

En lo personal, las motivaciones para la elección de este proyecto fueron la oportunidad de profundizar en los algoritmos de simulación estocásticos, y ver de una manera más extensa cómo obtener resultados similares utilizando métodos muy diferentes, como son la simulación estocástica exacta y aproximada. A su vez, y dado que trabajo como administrador de sistemas Unix-Linux desde hace más de 8 años donde prácticamente todo se realiza a través de comandos y terminales, me pareció muy interesante la idea de desarrollar una línea de comandos (*Command Line Interface* o CLI) de una aplicación como PeabraiN, que hasta ahora sólo ofrece una interfaz gráfica al usuario (excepto para el módulo de simulación). De esta forma se mejorará la forma de realizar los cálculos de una manera en cierto modo desatendida, obteniendo los resultados de una sola vez cuando han finalizado.

Por último, y dado que mi anterior proyecto fin de carrera de Ingeniería Técnica en Informática de Sistemas estuvo relacionado con contenidos dinámicos en la web a través de PHP4, me pareció atractivo el poder profundizar sobre la tecnología Ajax, que por aquel entonces no existía y que a día de hoy permite crear páginas web interactivas que se ejecutan en el cliente mientras se mantiene una comunicación asíncrona con el servidor.

1.3. Objetivo

Dada la diversidad de mejoras que se puede realizar a la aplicación PeabraiN, se han estudiado cuáles serían las más interesantes y cuáles podrían añadir una mayor funcionalidad a la herramienta. Es por eso que el objetivo de este PFC se centra en mejorar la herramienta de las carencias visibles más importantes, que se corresponden con las tres primeras fases del proyecto:

- **Selector de solucionadores de programación lineal:** Realizar un estudio en el código fuente de la aplicación Pipe2 [DKS, Pip], aplicación de creación y análisis de Redes de Petri a través de la cual funciona PeabraiN, y ver dónde y de qué manera añadir una opción que permita seleccionar los diferentes solucionadores de progra-

mación lineal que haya instalados el sistema. Además, buscar qué solucionadores existen actualmente en la red y ver cómo integrar todos ellos en la herramienta de manera que añadir o editar los solucionadores disponibles sea algo fácil de realizar para un usuario de la herramienta (no desarrollador).

- **CLI (*Command Line Interface*):** Realizar un estudio de los diferentes módulos que forman PeabraiN y ver qué tienen en común para poder desarrollar un CLI utilizando patrones de Software, desarrollándolo de manera que añadir un nuevo módulo al CLI sea una tarea rápida y sencilla para un usuario desarrollador medio, y además, que no sea necesario modificar el código fuente del CLI.
- **PeabraiN Web Interfaz:** Analizar de qué forma se puede crear un interface web de PeabraiN para que se pueda utilizar la herramienta a través de Internet, e implementar una prueba de concepto.
- **Simulación Estocástica Aproximada:** Analizar el simulador estocástico exacto que proporciona PeabraiN y que implementa el *Método de la primera reacción* [DKS, RJM12]. Estudiar otros métodos diferentes de simulación que mejoren el tiempo de simulación actual, e implementar alguno de ellos.

1.4. Métodos y herramientas empleadas

Durante el desarrollo del proyecto se han utilizado diferentes herramientas y estándares, como son:

- **UML**: El lenguaje unificado de modelado (Unified Modeling Language, UML) [Gro11] es un lenguaje que se emplea para especificar, documentar y visualizar modelos de sistemas software en forma de diagramas.
- **Redes de Petri**: [Mur89] Lenguaje de modelado formal que se utiliza para el modelado de sistemas concurrentes y distribuidos.
- **Pipe2**: [DKS, Pip] Herramienta desarrollada en Java para la creación y análisis de modelos estocásticos de Redes de Petri generalizadas (GSPN) [Cam98].
- **PeabraiN**: [RJM12] Conjunto de módulos basados en Pipe2 para cálculo de propiedades en Redes de Petri mediante problemas de programación lineal y simulación de Redes estocásticas.
- **Java**: Lenguaje de programación multiplataforma y orientado a objetos (versión 1.7 Update 55).
- **Eclipse**: Es una plataforma de desarrollo de código abierto basada en Java.
- **XMI (XML Metadata Interchange)**: [OMG07] Se trata de un lenguaje de especificación para el intercambio de datos entre herramientas. Está basado en el estándar XML, un estándar ISO.

- **Ajax (Asynchronous JavaScript And XML)**: [Hol08] Método de desarrollo web para crear aplicaciones interactivas que se ejecutan directamente en el cliente manteniendo una comunicación en segundo plano asíncrona con el servidor.
- **Visual Paradigm for UML**: Herramienta para la creación de diagramas UML.

1.5. Fases del trabajo

Una vez marcados los objetivos del proyecto, se organizó el esfuerzo de manera lógica para las diferentes partes de que constaba. Las principales etapas de esta labor fueron:

1. **Adquisición de conocimientos necesarios.**
   En primer lugar se adquirieron los conocimientos teóricos que iban a ser necesarios a lo largo del proyecto, como por ejemplo la profundización en Redes de Petri.
2. **Estudio de la Herramienta, Documentación y posibles mejoras.**
   En esta fase se analizó la herramienta PeabraiN, y entre todas las posibles mejoras que se podían realizar se seleccionaron las que mayor funcionalidad podían aportar a la misma. Posteriormente se procedió al estudio del código fuente de PeabraiN [RJM12] y Pipe2 [DKS, Pip], puesto que las mejoras implicaban la modificación de algunas partes de su código.
3. **Comprensión de la simulación estocástica de Redes de Petri.**
   Para poder implementar un método de simulación diferente es necesario comprender el simulador estocástico que viene dentro de la herramienta PeabraiN y cómo se utiliza con las Redes de Petri.
4. **Implementación de las mejoras y simulador. Pruebas.**
   Una vez terminada la recopilación de datos de la herramienta PeabraiN y del simulador estocástico exacto se realizaron la implementaciones de las diferentes mejoras de PeabraiN y del nuevo simulador, así como diferentes pruebas de todo lo implementado.
5. **Documentación final.**
   Finalmente, después de finalizar todas las fases se procede a realizar esta memoria con la documentación que se ha ido generando durante la vida del proyecto.

Capítulo 2

Conceptos previos

En este capítulo se pone en contexto al lector para que pueda entender de una manera más clara el resto de la memoria. Concretamente, se explica UML, patrones de diseño y Redes de Petri.

2.1. UML

El estándar UML es una especificación semi-formal que define un lenguaje gráfico que sirve para visualizar, especificar, construir y documentar los elementos de un sistema. Fue adoptado por Object Management Group (OMG) en 1998 y es un estándar ISO. Actualmente se encuentra en su versión 2.4.1 [Gro11]. Proporciona soporte para la planificación y control del ciclo completo de vida del software, independientemente de la plataforma para la que se desarrolla.

UML 2.4.1 define trece tipos distintos de diagramas gráficos que sirven para describir las vistas que un modelo puede necesitar para ser caracterizado, enfocados desde el paradigma Orientado a Objetos (OO). Para enumerarlos es mejor categorizarlos jerárquicamente: **diagramas de estructura** (diagrama de clases, componentes, objetos, estructura compuesta, despliegue y paquetes), **diagramas de comportamiento** (diagrama de actividades, casos de uso y estados) y por último los **diagramas de interacción** (secuencia, comunicación, tiempos y de vista de interacción). Los más empleados en este proyecto han sido *diagrama de clases* y *diagrama de secuencia*. A continuación se describen estos diagramas más en detalle.

2.1.1. Diagrama de clases

Un diagrama de clases está compuesto por *clases* y *relaciones*. Estos diagramas sirven para visualizar las relaciones entre las diferentes clases que involucran el sistema. La Clase es la unidad básica que encapsula toda la información de un Objeto (instancia concreta de una clase) y se representa por un rectángulo que posee tres divisiones. La división superior contiene el nombre de la clase, la división intermedia contiene los atributos que caracterizan la clase y la división inferior contiene los métodos u operacio-

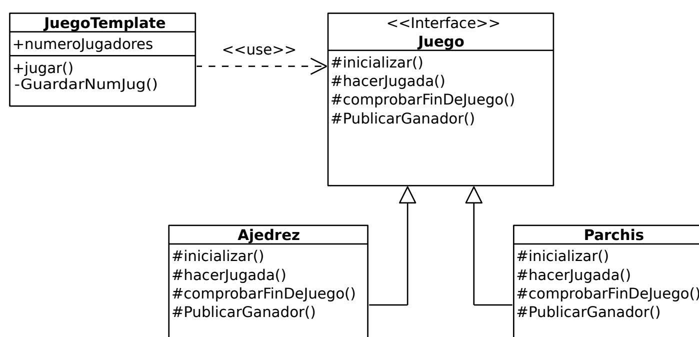

Figura 2.1: Ejemplo de Diagrama de clases UML.

nes, los cuales son la forma como interactúa el objeto con su entorno. Tanto la división intermedia como la división inferior son opcionales mientras que la división superior es obligatoria. Así mismo las clases se interrelacionan entre sí a través de los diferentes tipos de relaciones que existen, como por ejemplo Herencia, Composición, Agregación, Asociación y Uso.

En la Figura 2.1 se muestra un Diagrama de clases UML que muestra un diagrama básico para un interfaz de juegos. Está formado por tres clases (*JuegoTemplate, Ajedrez y Parchis*) y una interface (*Juego*), donde la clase *JuegoTemplate* tiene un atributo (+numerojugadores) y dos métodos métodos (+jugar y -GuardarNumJug). El resto de clases sólo tienen métodos. El símbolo + delante de los atributos o métodos significa que tiene un ámbito público, es decir, que puede ser accedido desde fuera de la propia clase. El símbolo - indica que es privado, por lo que los métodos o atributos sólo pueden ser accesibles desde dentro de la misma clase. Por último existe un tercer símbolo # que indica que es de ámbito protegido y se puede acceder a él desde las subclases, aún sin estar dentro del mismo paquete. Respecto a las relaciones, las clases *Parchis* y *Ajedrez* son especializaciones de herencia de la clase *Juego*, y a su vez la clase *JuegoTemplate* mantiene una dependencia o instanciación de uso de la clase *Juego*.

2.1.2. Diagrama de secuencia

En un diagrama de secuencia se muestran los módulos o clases que forman parte del programa así como las llamadas que se hacen en cada uno de ellos para realizar una tarea determinada. Se realizan diagramas de secuencia para definir acciones que se pueden realizar en la aplicación en cuestión. Un diagrama de secuencia contiene objetos, mensajes y líneas de vida. Los objetos se representan usualmente con rectángulos, los mensajes se representan con líneas continuas con una punta de flecha en un extremo, y el tiempo se representa en forma una progresión vertical, es decir, que lo que está más arriba en el diagrama es algo que sucede antes que lo que está representado más abajo. Los mensajes pueden ir desde la línea de vida de un objeto a otro (Ej. mensaje de *traspaso(cantidad)* en Figura 2.2), o al mismo objeto desde el que sale (Ej. mensaje de

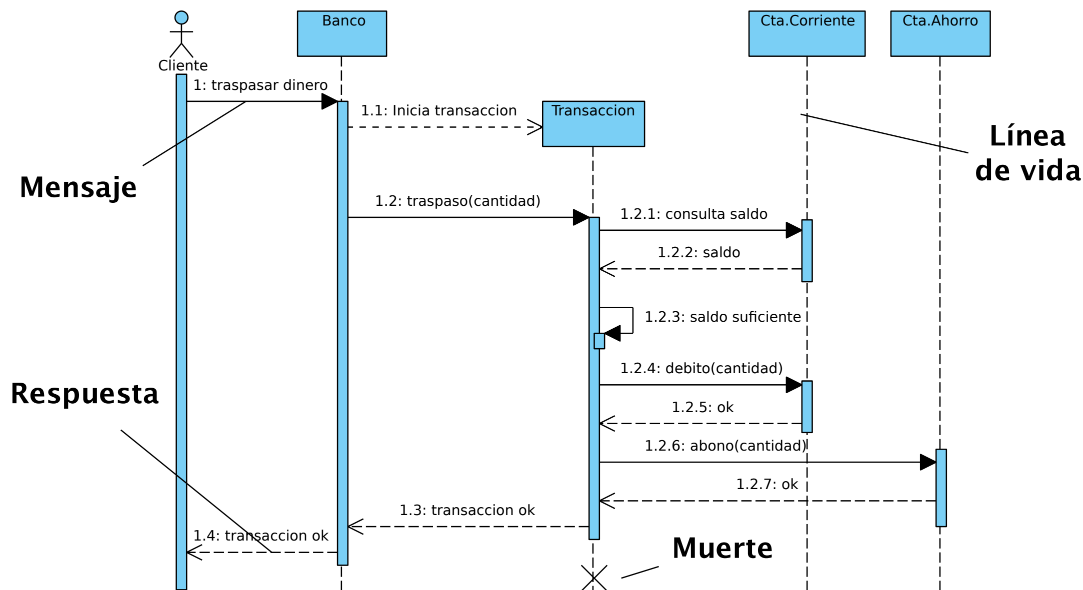

Figura 2.2: Ejemplo de diagrama de secuencia UML.

saldo suficiente en Figura 2.2), es decir, de su línea de vida a su propia línea de vida. Los mensajes a su vez pueden ser simples, síncronos o asíncronos. Un mensaje simple es la transferencia del control de un objeto a otro. Un mensaje síncrono es aquel en el que el objeto espera la respuesta a ese mensaje antes de continuar con su trabajo. Por último, un mensaje asíncrono es aquel en el que el objeto no espera a la respuesta a ese mensaje antes de continuar.

La Figura 2.2 muestra el diagrama de secuencia de una transacción bancaria, desde que se decide realizar un traspaso de dinero hasta que la transacción se ha realizado. Por ejemplo, en el caso de una aplicación de banca se podría realizar un diagrama de secuencia para "Realizar una transacción" o para acciones más específicas como podría ser "mover dinero".

2.2. Patrones de diseño

Los patrones son formas reconocidas que sirven para resolver de forma correcta diferentes problemas comunes a algunos escenarios. Para que una solución a alguno de estos escenarios pueda ser reconocida como patrón debe reunir ciertas características, que son: *efectividad* y *reusabilidd*. Para que un patrón sea efectivo debe haber sido verificado resolviendo situaciones iguales en ocasiones anteriores, y a su vez, para que sea reusable ha de poderse aplicar al diseño de diferentes problemas en diferentes circunstancias. Los patrones permitirían que algunos elementos de una estructura puedan cambiar independientemente del resto de elementos que forman esa estructura.

Existen multitud de tipos de patrones, pero los más conocidos son:

- **Patrones Creacionales.**
  Son aquellos que tratan de solucionar los problemas que surgen a la hora de crear objetos. Los patrones creacionales son: Object Pool, Abstract Factory, Builder, Factory Method, Prototipe, Singleton y Model View Controler (MVC).

- **Patrones Estructurales.**
  Este tipo de patrones crea grupos de objetos que ayudan a realizar tareas más complejas. Los patrones estructurales son: Wrapper, Bridge, Composite, Decorator, Facade, Flywight, Proxy, Module.

- **Patrones de Comportamiento.**
  Los patrones de comportamiento permiten precisar la comunicación entre los objetos de una aplicación así como el flujo de información que circula entre ellos. Los patrones de comportamiento son: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method y Visitor.

Para este proyecto se han utilizado dos patrones creacionales (*Model-View-Controller*, *Singleton*), y un patrón estructural (*Facade*), todos ellos se describen a continuación en más detalle.

Model-View-Controller (MVC)

El *Model-View-Controller* es un patrón que define una aplicación separándola en tres partes diferenciadas: el modelo (*Model*), la vista (*View*) y el controlador (*Controller*).

El *Model* es el encargado de representar los datos con los que la aplicación trabaja. Responde a peticiones realizadas por la vista y el controlador, pero no suele tener conocimiento de la existencia de los otros dos componentes.

La *View* recoge la información que le facilita el modelo y la muestra al usuario de forma que pueda ser interpretada y manejada por el mismo, normalmente a través de un interfaz gráfico. También contiene la lógica de interactuación con los dispositivos que conforman la interfaz de usuario (teclados, ratones, pantallas táctiles, etc..) generando los eventos necesarios que serán recibidos por el controlador.

El *Controller* recibe los eventos que genera el usuario a través de la vista, los procesa y genera las respuestas en el modelo. Algunas veces la acción del usuario no provoca respuestas sobre el modelo, sino sobre la propia vista, así que en estos casos el controlador es el encargado de iniciar las rutinas adecuadas para que la vista refleje los cambios.

Singleton

El patrón *Singleton* es uno de los más comunes y utilizados dentro del desarrollo de software. También se le conoce como patrón de única instancia, y su objetivo se basa en limitar la creación de objetos que pertenecen a una clase, garantizando que sólo se tenga una instancia de la clase en el transcurso de la aplicación, teniendo así un único punto de acceso global al objeto creado.

Facade

El patrón *Facade* sirve para simplificar el sistema y reducir su complejidad a través de la división en subsistemas, proporcionando una interfaz (de nivel más alto) de acceso a estos. Su objetivo es reducir la comunicación y dependencias entre subsistemas y, de esta manera, conseguir un sistema más sencillo (desde el punto de vista del cliente).

2.3. Red de Petri

Una Red de Petri (RdP) [Mur89] es un lenguaje de modelado formal que se utiliza para modelar sistemas cuyos comportamientos dinámicos se caracterizan por la concurrencia, la sincronización, la exclusión mutua y los conflictos. La representación gráfica de una RdP se realiza mediante un grafo dirigido en el que sus nodos pueden ser **lugares** (representados como círculos) o **transiciones** (representadas con cajas), y las uniones entre estos elementos se realizan a través de **arcos** que pueden estar dirigidos de una transición a un lugar o de un lugar a una transición, pero nunca de un lugar a un lugar o de una transición a una transición, y pueden tener un peso determinado que se indica con un número al lado o encima del arco. Un arco que no está etiquetado con un número significa que tiene por defecto un peso de 1. Los lugares suelen describir estados del sistema mientras que las transiciones se pueden interpretar como los eventos que modifican un determinado estado del sistema. Los lugares pueden tener a su vez **marcas o tokens**, que son la parte dinámica de una Red de Petri. El número de tokens en cada lugar puede variar y son los que determinan la situación de la red en un momento dado.

El funcionamiento dinámico de las Redes de Petri está dirigido por una regla de disparo, donde una transición puede dispararse si todos los lugares que tiene de entrada contienen por lo menos tantas marcas o tokens como las que se indiquen en el arco que conecta el lugar con la transición (transición habilitada). Una vez que se dispara la transición se eliminan en el lugar que precede a la transición tantos tokens como indique los arcos de entrada y se generan nuevos tokens en los lugares de salida, tantos como indiquen los arcos que conectan la transición con los lugares de salida.

Por ejemplo la Figura 2.3 representa una Red de Petri con cuatro transiciones y cinco lugares, de los cuales uno de ellos (P1) tiene una marca o token y todos los arcos tienen un peso de 1, por lo que la transición T1 está habilitada dado que el lugar que le precede tiene un número igual o mayor al peso del arco que los une. Si se dispara la transición habilitada T1, el lugar P1 se quedaría con cero marcas y tanto P2 como P3 quedarían ambas marcadas con un token.

Para este PFC se utilizan Redes de Petri estocásticas generalizadas (GSPN) [AMBC+95] que se caracterizan entre otras cosas por tener transiciones de dos tipos, *temporizadas* o *inmediatas*. Las transiciones temporizadas se representan con un rectángulo blanco y su disparo, como indica su propio nombre, tarda un tiempo que sigue una distribución de probabilidad. En este proyecto se consideran que las transiciones temporizadas siguen una distribución exponencial. Las transiciones

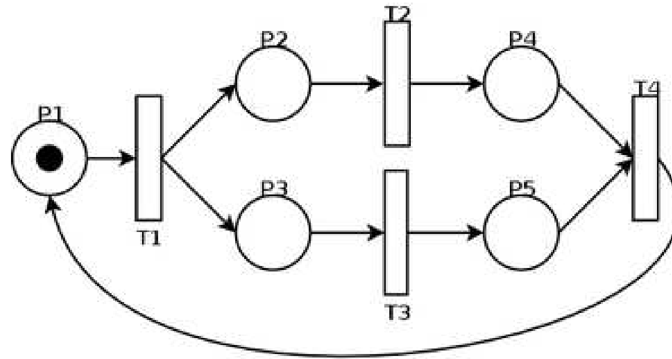

Figura 2.3: Ejemplo de Red de Petri.

inmediatas se representan con un rectángulo de color negro, y se disparan en un tiempo cero.

Capítulo 3

Mejoras en la funcionalidad de PeabraiN

En este capítulo se describe el trabajo llevado a cabo durante las tres primeras fases del proyecto. Antes de hablar en profundidad de PeabraiN, se necesita saber principalmente que PeabraiN es un conjunto de módulos que funcionan bajo otra herramienta diferente llamada Pipe2 [DKS, Pip]. PeabraiN depende completamente de Pipe2, y sus módulos suponen una adición a la funcionalidad proporcionada por los módulos nativos de Pipe2.

Pipe2 *(Platform-Independent Petri Net Editor 2)* [DKS, Pip] es una herramienta desarrollada en Java para la creación y análisis de Redes de Petri estocásticas generalizadas (GSPN). Fue creado por un grupo de desarrolladores como posgrado en el Departamento de Informática del *Imperial College* de Londres. Desde entonces ha sido constantemente mejorada hasta llegar a la versión 4.2.1, la cual se utilizará como base de este proyecto. Además de la creación y modificación de modelos de Red de Petri, Pipe2 posee la opción de poder añadir módulos externos, característica que la diferencia de otras muchas herramientas similares cuyo análisis funcional suele ser fijo y no puede ser aumentado por el usuario. Es por esto que Pipe2 proporciona un trampolín ideal para la experimentación de nuevas técnicas de análisis sobre Redes de Petri.

A raíz de esta primera aplicación y su característica de adición de módulos nace PeabraiN, acrónimo de *Performance Estimation bAsed (on) Bounds (and) Resource optimisAtIon (for Petri) Nets* (`http://webdiis.unizar.es/GISED/?q=tool/peabrain`). PeabraiN es un conjunto de módulos desarrollados por el grupo de investigación GISED que permiten, entre otras funcionalidades, realizar estimaciones de rendimiento, optimización de recursos, cálculo de cotas estructurales de marcado y habilitación, cálculo de las ratios de visita de transiciones y simulaciones de Redes de Petri estocásticas. Muchas de estas funcionalidades son calculadas a través de problemas de optimización lineal.

PeabraiN [RJM12], como se puede ver en la Figura 3.1 está formado por una arquitectura cerrada de capas, donde cada una de ellas sólo se comunica con las capas inmediatamente inferiores. La capa de datos (*Data layer*) está formada por las clases que tienen la información necesaria para los algoritmos que ejecutan los módulos, la capa intermedia (*Intermediate layer*) contiene las clases que implementan los algoritmos de

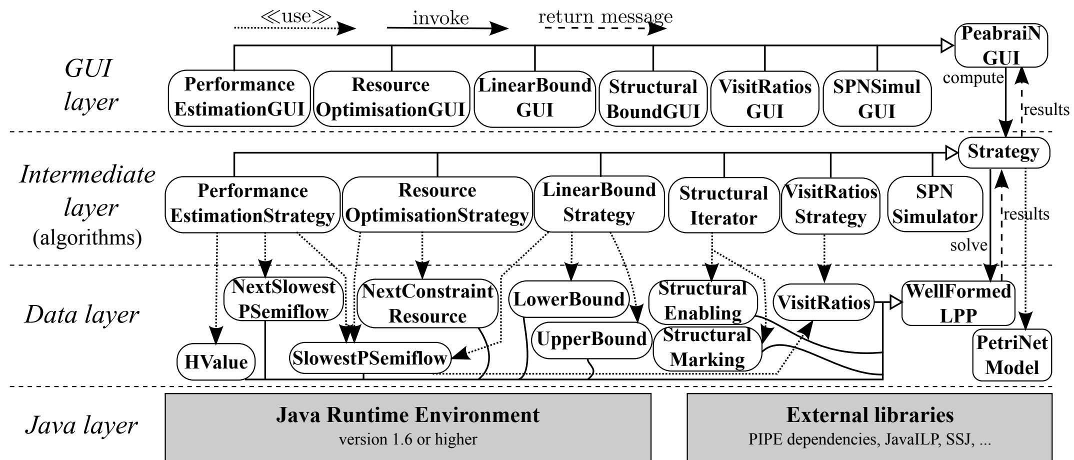

Figura 3.1: Arquitectura de software de PeabraiN (extraído de [RJM12]).

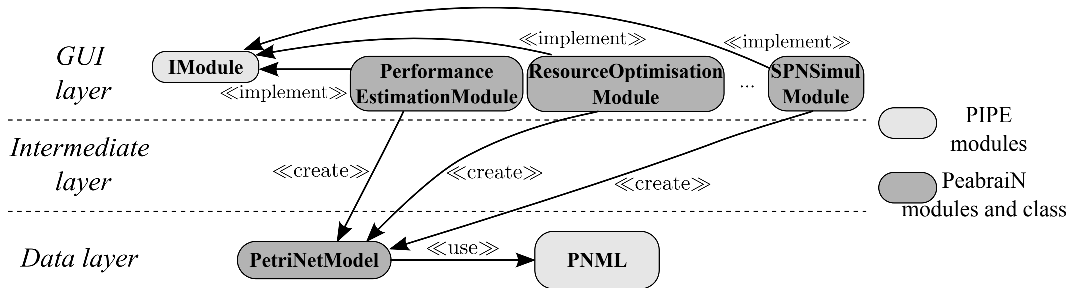

Figura 3.2: Integración de PeabraiN con Pipe2 (extraído de [RJM12]).

cada uno de los módulos y la capa GUI (*GUI layer*) contiene las clases que crean el interfaz gráfico que recoge los datos del usuario. Como se observa en la figura, cada una de estas capas coincide con el patrón de software *Model-View-Controller* (explicado en la Sección 2.2).

En la Figura 3.2 se puede observar cómo están relacionados PeabraiN y Pipe2. Puesto que Pipe2 es extensible a través de módulos, cada uno de esos módulos debe implementar la interfaz IModule.

Una vez mostrado qué son cada una de las herramientas y cómo se integran entre ellas se definen las 3 diferentes mejoras que se han desarrollado a lo largo de este PFC.

3.1. Selector de solucionadores de Programación Lineal

3.1.1. Análisis

Uno de los problemas observados después de analizar PeabraiN es que no hay implementada una opción que permita seleccionar un solucionador de Programación Lineal (PL). De hecho, Pipe2 no dispone ni tan siquiera de un menú de preferencias u opcio-

nes que permita al usuario configurar los aspectos generales. Como se ha descrito, esta carencia supone que los cálculos de los módulos de PeabraiN siempre van a ser resueltos por el solucionador que haya usado el desarrollador en el código, siendo en este caso GLPK (*GNU Linear Programming Kit*) [glp06]. Sin embargo, como se puede apreciar en la Tabla 3.1, existen multitud de solucionadores de PL disponibles en el mercado. Así pues, se considera una gran mejora el poder añadir alguno de estos solucionadores, o incluso añadir alguno que no esté en la Tabla 3.1.

Nombre	**URL**	**Licencia**
lpsolve	http://lpsolve.sourceforge.net/5.5/	libre bajo LGPL
ILOG CPLEX	http://www.ilog.com/products/cplex/	comercial
Gurobi	http://www.gurobi.com/	comercial
Mosek	http://www.mosek.com/	comercial
GLPK	http://www.gnu.org/software/glpk/	libre bajo GPL
SAT4J	http://sat4j.org/	libre bajo LGPL
MiniSAT+	http://minisat.se/MiniSat+.html	libre bajo MIT

Tabla 3.1: Solucionadores de Programación Lineal.

Esta mejora se ha añadido a PeabraiN y Pipe2 teniendo en cuenta dos objetivos principales:

1. Tratar de modificar el menor código posible en la herramienta Pipe2 para añadir un elemento que permita al usuario seleccionar un solucionador de manera sencilla.

2. Que el desarrollo de este selector de solucionadores no tenga que ser modificado a posteriori por un usuario no desarrollador que quiere modificar la lista de solucionadores disponibles en el selector.

Para cumplir el primer objetivo se decide añadir un menú de "Settings" o "Preferences" que es muy común en la mayoría de aplicaciones. Esta nueva opción es idónea porque tiene la ventaja de que sólo es necesario añadir una nueva acción a los menús de Pipe2, lo que implica modificar muy poco el código fuente original. Desde esta nueva acción "Settings" se puede mostrar una ventana cuyo código ya es totalmente independiente de Pipe2 y desde la que se pueden elegir los diferentes solucionadores.

Para el segundo objetivo se necesita encontrar una forma en que el usuario sea capaz de añadir, modificar o editar el listado de solucionadores sin modificar el código fuente de la aplicación, por lo que se ha de buscar un punto de unión entre el usuario y la herramienta que permita realizar esta operativa de manera sencilla. No se pretende gestionar (añadir, editar o modificar) la lista de solucionadores disponibles desde la nueva ventana puesto que ese listado no es algo que varíe con frecuencia, simplemente que el usuario pueda seleccionar un solucionador u otro desde la propia herramienta. Para solucionar este objetivo se ha optado por utilizar un fichero editable por el usuario que la herramienta es capaz de leer y mostrar los solucionadores en función del contenido de este fichero.

Figura 3.3: Opción de selección de solucionadores.

3.1.2. Diseño

La nueva ventana que se muestra en la Figura 3.3 permite seleccionar los diferentes solucionadores y además muestra si están o no instalados en el sistema, así como la ruta absoluta a la librería de cada uno de ellos. La inserción de esta nueva opción “Settings” a los menús de Pipe2 supone modificar ligeramente su código fuente, por lo que se proporcionan las pequeñas modificaciones realizadas en forma de ficheros “.patch” dado que es la opción más cómoda para activar esta funcionalidad en el caso de que aparezca una nueva versión de Pipe2 (siempre y cuando el código de creación de los menús de Pipe2 no sea modificado).

De todos los diferentes patrones explicados en la sección 2.2 hay dos que encajan en este diseño: *Model-View-Controller* y *Singleton*.

En el selector de solucionadores que se ha desarrollado en el PFC se puede observar claramente el patrón MVC utilizado en las tres clases que se usan para crear y gestionar la ventana de la Figura 3.3. Estas clases que forman el patrón MVC son **SolverSelectorGUI** (que se corresponde con la *View*), **SolverSelectorController** (que se corresponde con el *Controller* del patrón) y **SolverSelectorModel** (que se corresponde con el *Model*), todas ellas visibles en la Figura 3.4.

El patrón *Singleton* que se ha utilizado se muestra en la Figura 3.5 y sirve para asegurar que sólo se crea una única instancia del solucionador, ahorrando tiempo de ejecución y evitando que haya más de un solucionador en el caso de que se abran múltiples ventanas para realizar cálculos de forma paralela.

3.1.3. Configuración

Para que el usuario pueda modificar la lista de solucionadores de una forma sencilla y eficaz se va a implementar un fichero de configuración “.solverselector.conf” externo a la aplicación que se ubicará en la carpeta “Home” del usuario.

El fichero “.solverselector.conf” mostrado en la Figura 3.6 se crea por defecto la primera vez que el usuario ejecuta la acción de selección de solucionadores, y su contenido

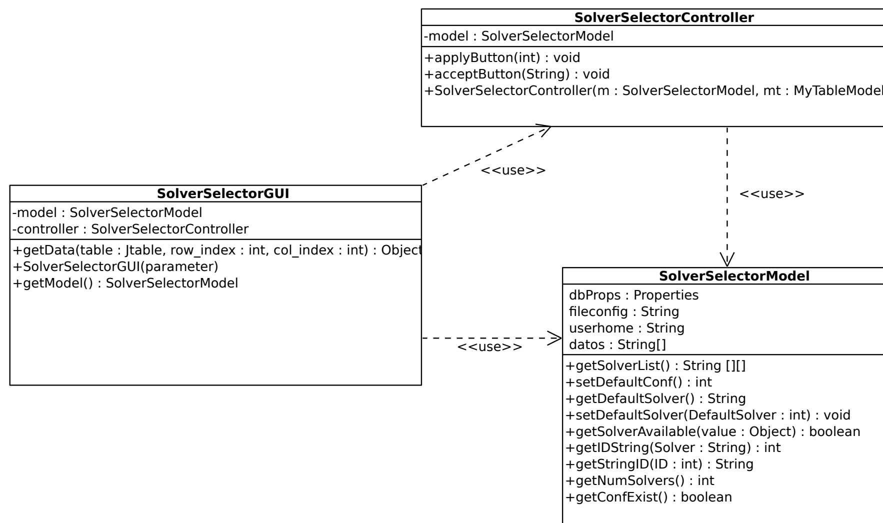

Figura 3.4: Diagrama de clases de selector de solucionadores.

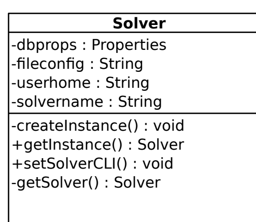

Figura 3.5: Diagrama de clases del patrón *Singleton*.

```
#Tue Apr 08 19:19:32 CEST 2014
Default = GLPK
Solver7 = MiniSat+,/usr/lib/MiniSat+
Solver6 = SAT4J,/usr/lib/SAT4J
Solver5 = GLPK,/usr/lib/libgllpk.so
Solver4 = Mosek,/usr/lib/Mosek
Solver3 = Gurobi,/usr/lib/Gurobi
Solver2 = ILOG_CPLEX,/usr/lib/ilog_cplex
Solver1 = Lp_solve,/usr/lib/lp_solve
```

Figura 3.6: Ejemplo de configuración del fichero ".solverselector.conf".

está formado por un seleccionador por defecto y una lista de los posibles solucionadores que se quiere aparezcan en la ventana de "Settings". En caso que se quiera añadir un nuevo solucionador basta con editar manualmente el fichero y añadir una nueva línea respetando el número de solucionador con dos campos separados por una coma, uno con el nombre del solucionador y otro con la ruta absoluta del solucionador dentro del sistema operativo. De esta manera, el contenido de la ventana de configuración mostrada en la Figura 3.3 se genera de forma dinámica en función del fichero ".solverselector.conf".

3.2. Línea de comandos (*Command Line Interface*, CLI)

La creación de un CLI para PeabraiN surge por la necesidad de lanzar varios cálculos de forma desatendida. En la actualidad, si sólo se dispone del interfaz gráfico y se quiere realizar 100 cálculos, ¿qué ocurre? que el usuario debe realizar varias acciones: cargar la Red de Petri, seleccionar el tipo de cálculo, configurar los parámetros, ejecutar el cálculo, esperar a que éste finalice y guardar los resultados, y todo esto además multiplicado por el número de cálculos que se quieren realizar. Esta necesidad de disponer de un CLI se incrementa además si se tiene en cuenta que al concluir la primera fase de este proyecto ya se pueden realizar cálculos con diferentes solucionadores, lo que habilita el poder enviar cálculos de una misma Red de Petri utilizando diferentes solucionadores para por ejemplo poder contrastar después los datos obtenidos (es decir, evaluar diferentes solucionadores). Las posibilidad de realizar cálculos por lotes de forma desatendida es seguramente una de las mayores mejoras que se puede realizar a PeabraiN.

El objetivo para desarrollar el CLI de PeabraiN pasa por ser totalmente independiente de la aplicación visual, pero a su vez debe utilizar el código de PeabraiN para realizar los cálculos de PL.

El CLI ha sido desarrollado utilizando el patrón de diseño *Facade* mostrado en la Figura 3.7 para permitir la inclusión de nuevos módulos de PeabraiN de forma flexible y sencilla. En este caso se utiliza este patrón para solucionar un problema de dependencia entre el código que proporciona el CLI y el código que es individual para cada módulo, ya que cada nuevo módulo debe utilizar los métodos abstractos indicados por la clase

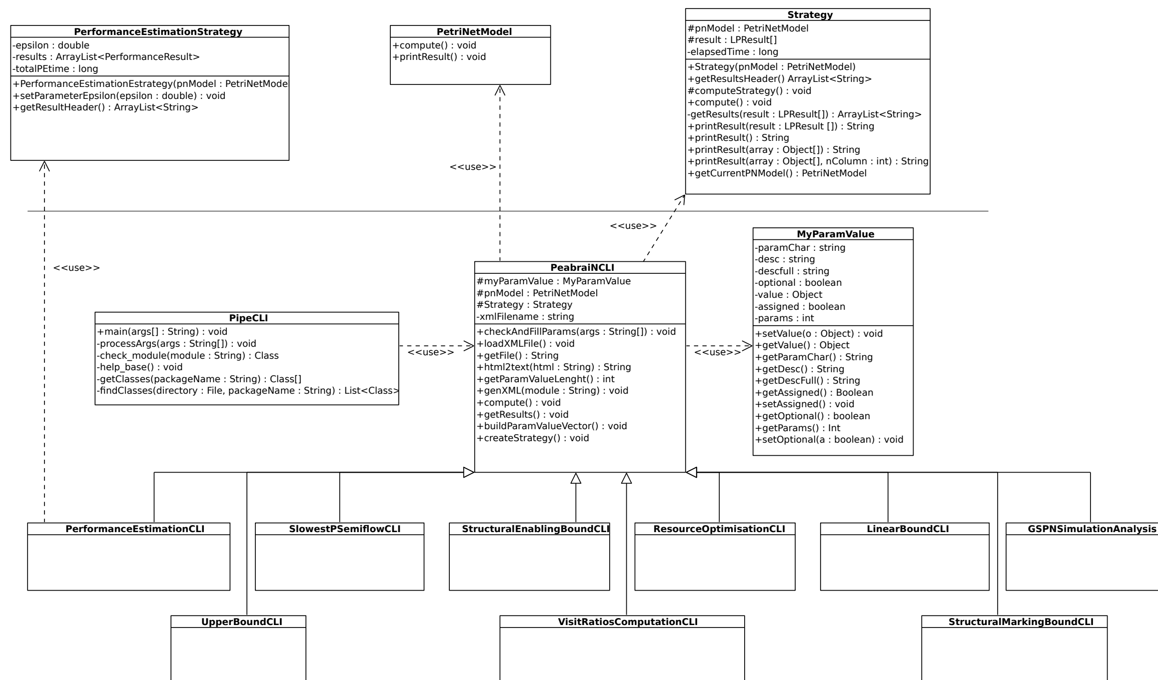

Figura 3.7: Diagrama de clases UML del CLI (*Command Line Interface*).

PeabraiNCLI, *buildParamValueVector()* y *createStrategy()*.

En el diagrama de secuencia de la Figura 3.8 se observa el flujo seguido por la arquitectura desde que el usuario quiere realizar un cálculo hasta que éste es devuelto por *PeabrainCLI*. En primer lugar, el usuario introduce la llamada a la clase *PipeCLI* con los parámetros adecuados para el módulo que quiere utilizar, y si estos parámetros no son suficientes se devuelve un mensaje de ayuda base mostrando el funcionamiento correcto del CLI. Si los parámetros son suficientes se transfieren a la clase *PeabrainCLI* que será la encargada de verificar que todos son correctos en función del módulo que se quiere utilizar, y si esto es así se llama al método *compute()*. Este método carga la Red de Petri que está en formato XML, establece los parámetros de la estrategia a través de la clase específica del módulo y finalmente instancia la estrategia que es la que realiza los cálculos. Una vez finaliza devuelve los resultados obtenidos y es PipeCLI a través del método *getResult()* el que se encarga de mostrarlos por pantalla.

Al desarrollar esta fase del proyecto se ha tenido muy en cuenta el poder añadir nuevos módulos a PeabraiN y cómo facilitar al máximo la integración de un nuevo módulo en el CLI de forma que no haya que modificar el código base de PipeCLI. Es por esto que el diseño utilizado tiene una serie de ventajas que permite que al usuario añadir un nuevo módulo con tan solo crear una pequeña clase en una ruta específica (<Pipe>/modules/CLI) con los 2 métodos indicados anteriormente *buildParamValueVector()* y *createStrategy()*. La otra ventaja principal es que la ayuda generada para cada módulo es dinámica y el usuario no tiene que configurar nada más para que PipeCLI

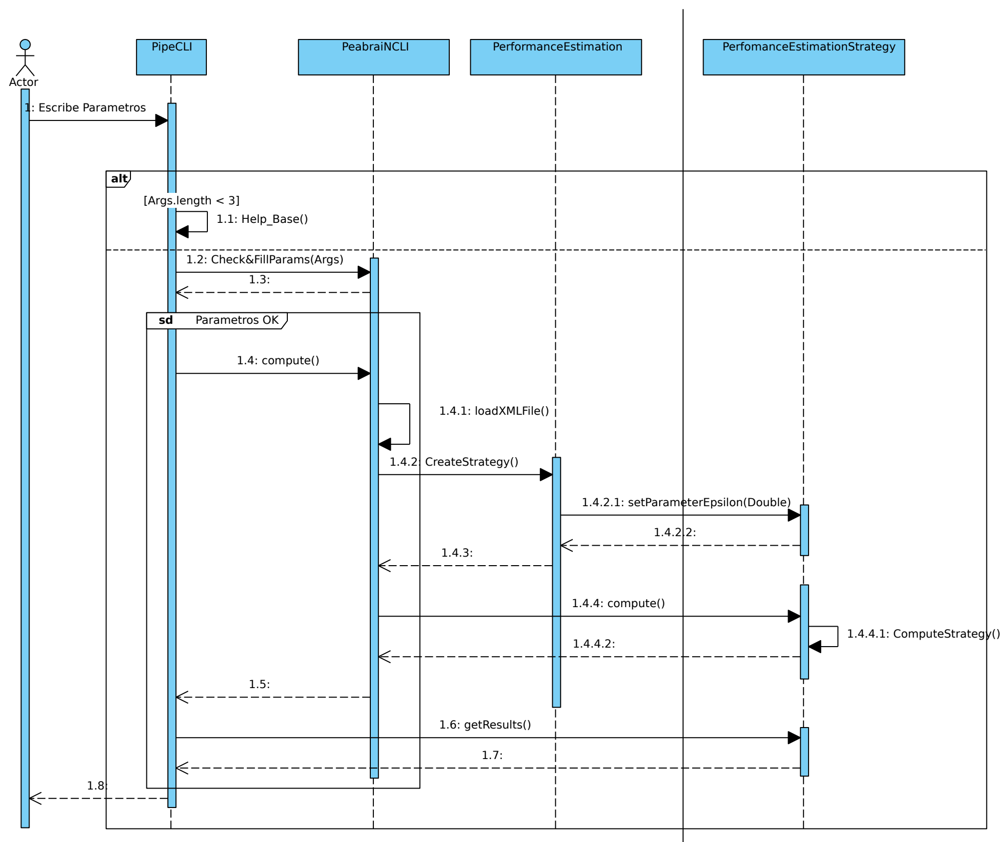

Figura 3.8: Diagrama de secuencia UML del CLI.

```
user@ubuntu:~/PeabraiN421/src$ ./PipeCLI -help
Use: PipeCLI -m <module> -h

------ AVAILABLE MODULES ------
- ResourceOptimisation
- PerformanceEstimation
- SlowestPSemiflow
- UpperBound
- LowerBound
- VisitRatiosComputation
- StructuralEnablingBound
- StructuralMarkingBound
```

Figura 3.9: Ejemplo de ayuda para listar los módulos disponibles en PeabraiN a través del CLI.

pueda trabajar con el nuevo módulo. En la Figura 3.9 se muestra la ayuda dinámica generada por PeabrainCLI cuando se busca información de los módulos que hay disponibles. El listado de "AVAILABLE MODULES" se genera en función de los módulos disponibles en PeabraiN.

Por otra parte, PipeCLI también es capaz de generar dinámicamente la ayuda de los parámetros que tiene cada módulo. Estos parámetros tienen una serie de propiedades que gracias a la clase *MyParamValue*, utilizada en la arquitectura, permite añadir nuevas propiedades a un parámetro si fuera necesario. La clase *MyParamValue* posibilita de una manera muy sencilla añadir nuevos parámetros a un módulo sin tener que modificar el resto de clases. Los atributos de esta clase se muestran en la Figura 3.10 (inicializados en el constructor) y permiten observar los diferentes elementos que se necesitan para conformar un parámetro: "paramChar" guarda el parámetro (por ejemplo *-f*); "desc" almacena una breve descripción del parámetro y "descfull" almacena una descripción más ampliada, ambas utilizadas para la ayuda dinámica generada en cada módulo. "optional" se utiliza para indicar si un parámetro es obligatorio o no al utilizar un módulo. "value" es el valor del parámetro; "assigned" se usa para saber si un parámetro requiere de un valor o únicamente es un parámetro indicativo; por último "params" es un valor numérico que se utiliza para controlar los parámetro pasados al módulo desde el CLI y poder determinar si el uso es correcto.

Existen 2 tipos de parámetros: los parámetros que son comunes a todos los módulos de PipeCLI y los parámetros que son específicos de cada módulo, y en base a estos dos tipos de parámetros se genera la ayuda del módulo como se muestra en la Figura 3.11. Son parámetros comunes el que indica el fichero que contiene la RdP en formato XML ("-f"), el que indica el solucionador que se quiere utilizar para realizar los cálculos ("-solver") y por último el parámetro que especifica si el formato de salida es en texto plano en formato HTML ("-fancy-output"), posteriormente utilizado para la mejora web.

```
public class MyParamValue {
        private String paramChar;
        private String desc;
        private String descfull;
        private boolean optional;
        private Object value;
        private boolean assigned;
        private int params;

        }
```

Figura 3.10: Atributos de la clase *MyParamValue*.

```
user@ubuntu:~/PeabraiN421/src$ ./PipeCLI -m PerformanceEstimation -h
Module name: PerformanceEstimation
=========================
Module Parameters:
 -m <module name>
 -f <PetriNetFile.xml>
 -solver <Solver>
 -fancy-output // Dump output in html format
 -e <epsilon> // Epsilon value
=========================
```

Figura 3.11: Ejemplo de ayuda para el módulo *PerformanceEstimation*.

En la Figura 3.11 se muestra el resultado de solicitar la ayuda para un módulo específico de PeabraiN a través del CLI. Los parámetros mostrados son todos los comunes, y además el parámetro "-e", que corresponde a un parámetro específico del módulo *PerformanceEstimation*. Los parámetros específicos de cada módulo se configuran dentro del método *buildParamValueVector()* que está en la clase del propio módulo. Para que el CLI pueda entender los diferentes parámetros es necesario configurar cada uno de ellos con unos datos básicos y obligatorios, que son: el tipo de parámetro, el nombre, la descripción, si es un parámetro obligatorio u opcional, si es un parámetro que requiera un valor, y en este caso el tipo de valor esperado.

Considérese un ejemplo donde se quiere añadir un nuevo parámetro específico "*value*" al módulo ya existente *PerformanceEstimation*. En el método *buildParamValueVector()* se debe añadir el siguiente código fuente (explicado a continuación):

```
myParamValue[3] =
  new MyParamValue("-v","value","full description",
```

```
                    false,new Double(0.0),true,2);
```

- **-v**: parámetro utilizado en el terminal.
- **value**: Nombre del parámetro.
- **full description**: Descripción del parámetro (para la ayuda).
- **false**: Indica que el parámetro no es obligatorio para este módulo.
- **new Double(0.0)**: Tipo del parámetro.
- **true**: Indica que el parámetro requiere un valor posterior.
- **2**: Indica el número de valores necesarios para ese parámetro (Parámetro + valor).

Después de añadir esta línea dentro del código fuente del módulo *PerformanceEstimation* el resultado sería que al utilizarlo se permitiría usar el nuevo parámetro "*-v*" de manera "*opcional*" y que requeriría un número de tipo "*Double*". Ejemplo de como habría que utilizar el módulo con el nuevo parámetro añadido:

```
#PipeCLI -m PerformanceEstimation -e 3.0 -v 43.2
```

3.3. Cliente Web

Esta tercera mejora de la aplicación surge de la necesidad de poder realizar cálculos de PeabraiN a personas que no lo tengan instalado y que dispongan de una conexión a Internet. Para presentar la herramienta a través de Internet se pensó en crear una aplicación en Java que conectaría el servidor web y PeabraiN, pero tenía la desventaja de mantener una aplicación Java constantemente corriendo en el servidor y además requería de un trabajo de desarrollo considerable. Posteriormente se pensó en reutilizar la segunda mejora de este proyecto, el CLI, de manera que sólo es necesario desarrollar una capa web que utilice el CLI para realizar los cálculos y posteriormente los muestre a través de la web. Con esta segunda opción el desarrollo se simplifica considerablemente manteniendo el objetivo de usar la herramienta a través de Internet.

Para simplificar al máximo el uso de la la aplicación se diseña con una única pantalla que se muestra en la Figura 3.12 donde pretende que aparezcan todas las opciones necesarias para realizar un cálculo cualquiera. Este modelo de una sola ventana es posible gracias al uso de Ajax [Hol08], dado que los datos se solicitan en segundo plano al servidor y se muestran de forma instantánea sin necesidad de recargar la página completa, mejorando la interactividad con el usuario, así como la velocidad de respuesta de la aplicación.

En primer lugar el usuario debe subir la Red de Petri sobre la que se quieren realizar los cálculos en formato PNML al servidor a través de la ventana central. Una vez el servidor recibe la red la procesa y la muestra en forma de imagen igual que se vería en la herramienta Pipe2. Después el usuario ha de elegir el tipo de cálculo que se quiere

Figura 3.12: Diseño de la aplicación PeabrainWEB de una única ventana.

PEABRAIN WEB

Upload Your PetriNet .xml file

Seleccionar archivo shop.xml

Successfully Uploaded file

Available Modules
Resource Optimisation
Performance Estimation
Upper Bound
Lower Bound
SlowestPSemiflow
Visit Ratios Computations
Structural Enabling Bound
Structural Marking Bound

Available Solvers
GLPK
Gurobi
ILOG_CPLEX
Lp_solve
MiniSat+
Mosek
SAT4J

Variable module = PerformanceEstimation
epsilon: 333
Submit

fancy-output -solver GLPK -e 333

ANSWER:
Checking module... Loading... Done

Regrowing step	Components	Throughput	Error accuracy	Sim. time
0	{P0, P1, P3, P4, P6, P7, P8, P9}	0.384612	?0.0000	0.0010

Total elapsed time: 0.0000 seconds.

Figura 3.13: Imagen de la aplicación PeabraiN Web. Resultado final.

realizar a través de los diferentes módulos disponibles, y en función de estos la interfaz mostrará los parámetros requeridos para este módulo.

La comunicación de parámetros entre PipeCLI y PeabraiN Web se realiza a través de XML. Cuando un usuario desde el CLI o PHP realiza una solicitud de ayuda para un módulo, PipeCLI además de mostrarla en el terminal genera un fichero XML con todos los parámetros e información necesaria que posteriormente será recogido por PeabraiN Web para mostrar los campos necesarios de inserción de parámetros en el interfaz web. De esta forma, un usuario que quiera insertar su propio módulo en PeabraiN no tiene que modificar nada del código Web, es decir, en el momento que el nuevo módulo se pueda utilizar a través del CLI automáticamente se podrá utilizar desde la Web.

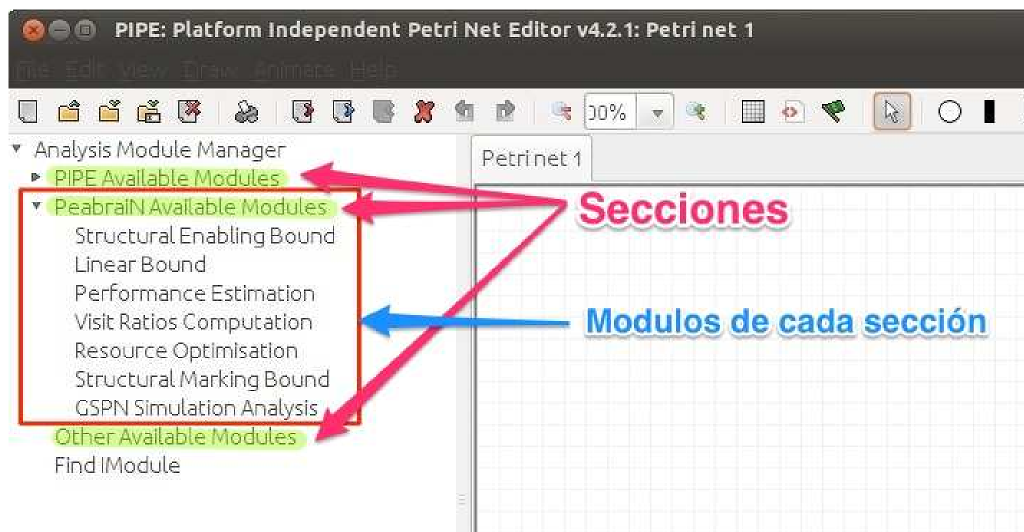

Figura 3.14: Categorización de la lista de módulos disponibles en Pipe2.

3.4. Extras

Además de las tres primeras fases realizadas hasta ahora, y para poder diferenciar visualmente los módulos que vienen por defecto con Pipe2 de los módulos que proporciona PeabraiN, se ha modificado el código de Pipe2 de manera que ahora se agrupan todos los módulos de Pipe2 en una misma sección y todos los módulos de PeabraiN en otra sección diferente, tal y como se puede ver en la Figura 3.14.

3.5. Código fuente

Todas estas mejoras se han desarrollado con licencia GPL y se encuentras disponibles en la página de Internet:

`https://bitbucket.org/rjrodriguez/peabrain/`

Capítulo 4

Mejoras en la computación de PeabraiN

En este capítulo se analiza el algoritmo de simulación estocástica exacta que proporciona la herramienta PeabraiN, y a su vez se busca la manera de mejorarlo. Así pues, se implementa un simulador estocástico aproximado aplicado a Redes de Petri, basado en el método de la primera reacción [Gil76], utilizado principalmente en simulación de reacciones químicas. En la actualidad existen diferentes métodos de simulación además de los dos que formuló Gillespie en 1977 (*Método directo* y *Método de la primera reacción*), mostrados en la Tabla 4.1 junto con su año de publicación. Los métodos más lentos se muestran en la parte inferior mientras que los métodos más rápidos se encuentran en la parte superior de la tabla. Esta forma de exponer los diferentes métodos permite ver claramente que los métodos aproximados son más rápidos que cualquier otro método exacto, aunque los resultados sin embargo pueden no ser tan precisos.

MÁS RÁPIDO, MEJOR

Discretos, Exactos	**Continuos, Aproximados**
	Tau-Leaping Modificado (2005)
	Tau-Leaping (2001)
Método directo logarítmico (2006)	
Método directo de ordenación (2005)	
Método directo optimizado (2004)	
Método de la siguiente reacción (2000)	
Método directo (1977)	
Método de la primera reacción (1977)	

MÁS LENTO, PEOR

Tabla 4.1: Tabla de algoritmos de simulación estocástica.

4.1. Método de simulación estocástica actual

En primer lugar se analiza el método utilizado por el simulador que proporciona la herramienta PeabraiN, y se observa que está utilizando el método de la primera reacción [Gil76], que fue diseñado para simular la evolución temporal de un sistema (en este caso de una Red de Petri) de manera exacta. Los pasos que sigue son:

- **Inicialización**: Inicializa el número de moléculas en el sistema. En este caso, hablando en términos de Redes de Petri es lo mismo que inicializar la red disparando todas las transiciones habilitadas inmediatas hasta que sólo quedan transiciones temporizadas por disparar.
- **Monte Carlo**: Genera números aleatorios para determinar cuál será la próxima reacción química que ocurrirá, dando mayor probabilidad a aquellas reacciones que tienen más moléculas. En este caso de RdP es equivalente a calcular números aleatorios para determinar qué transición temporizada habilitada se disparará a continuación dando mayor probabilidad a las que tengan un menor tiempo de disparo.
- **Actualización**: Incrementa el tiempo que ha transcurrido en el paso anterior y aumenta el contador de disparos.
- **Iteración**: Regresa al segundo paso (Monte Carlo) a menos que no haya más reacciones disponibles (en este caso transiciones por disparar) o el tiempo de simulación haya sido excedido.

4.2. Nuevo método de simulación estocástica

El método de simulación que se utiliza como mejora en este PFC va a ser el método de “Tau-Leaping (2001)”. Es un método aproximado para la simulación de sistemas estocásticos basado en el algoritmo de Gillespie, y su funcionamiento radica en calcular tiempos pequeños de salto tau (τ) con los que ir evolucionando la Red de Petri, de manera que los cambios inducidos por estos saltos deben ser lo suficientemente pequeños para que se produzcan cambios sustanciales en el sistema y la Red vaya evolucionando de la forma más real posible, pero a su vez estos tiempos de salto no pueden ser demasiado grandes porque provocarían cambios demasiado bruscos en el sistema y haría que ésta no evolucionase de manera real, obteniendo así unos resultados erróneos. Este método, por su definición, puede generar en ocasiones poblaciones negativas en el sistema si τ es demasiado pequeño.

Los saltos que se van calculando en el transcurso de la simulación dependen de las transiciones habilitadas de la Red de Petri en el momento en que se calcula cada nuevo salto. Una vez se obtiene el siguiente salto se utiliza la distribución de Poisson para estimar el número de disparos que se realizan en función del tiempo de salto ya calculado.

La diferencia principal entre este método Tau-Leaping y el método directo radica en que este último es demasiado lento, ya que en cada paso de la simulación sólo dispara una

transición temporizada habilitada, mientras que Tau-Leaping dispara en cada salto todas las transiciones temporizadas habilitadas (en función de la distribución de Poisson).

```
1  Inicialización de variables;
2  si tiempo máximo de simulación no establecido por el usuario entonces
3  |  Establecer tiempo máximo
4  fin si
5  Se disparan todas las transiciones inmediatas habilitadas
6  Se calculan las transiciones temporizadas habilitadas
7  si tamaño de temporizadas habilitadas = 0 entonces
8  |  La solución no es posible. Chequear marcado inicial.
9  |  Salida.
10 fin si
11 Cálculo del salto tau
12 Se disparan temporizadas dependiendo del tau calculado
13 mientras tiempo de simulación < tiempo máximo hacer
14 |  Disparar todas las transiciones inmediatas habilitadas
15 |  Calculo del salto tau
16 |  Se disparan temporizadas dependiendo del tau calculado
17 |  Incrementar tiempo simulación salto tau
18 fin mientras
```

Algoritmo 1: Pseudocódigo simulación estocástica aproximado. Método Tau-Leaping.

El pseudocódigo del método Tau-Leaping [Gil07, Gil08] se muestra en el Algoritmo 1. Inicialmente, se disparan todas las transiciones inmediatas habilitadas *(Línea 5)* y posteriormente se calcula la primera condición de salto τ como se explica a posteriori. En este punto es cuando se recorre una única vez la Red de Petri buscando las transiciones temporizadas habilitadas *(Línea 6)* y calculando un salto τ en función de estas transiciones *(Línea 11)*. Una vez calculado el tiempo de salto se disparan todas las transiciones habilitadas un número de veces calculado como el resultado de la distribución inversa de Poisson según la fórmula:

$$n^{o} disparos = \mathcal{P}_i(a_j(x)\tau) \tag{4.1}$$

donde $\mathcal{P}_i$ es la distribución de Poisson de $a_j(x)\tau$, y $a_j(x)$ es la ratio de la transición por su "enablingdegree" multiplicado por τ. El *enablingdegree* de una transición es el menor valor del conjunto de lugares que la preceden multiplicado por el peso del arco que las une.

Éste es en términos generales el funcionamiento del método Tau-Leaping, que se repite constantemente dentro del bucle del algoritmo *(Línea 13 a Línea 18)* hasta que el tiempo de simulación real de la Red de Petri alcanza un tiempo máximo establecido por el usuario al inicio de la simulación.

El cálculo de τ [Gil07, Gil08] se deduce de la siguiente formula:

$$\tau = \min_i \left\{ \frac{max\{\epsilon_i x_i, 1\}}{\left|\sum_j v_{ij} a_j(x)\right|}, \frac{max\{\epsilon_i x_i, 1\}^2}{\sum_j v_{ij}^2 a_j(x)} \right\} \tag{4.2}$$

Como ya se ha comentado anteriormente, esta fórmula se originó para simular de forma estocástica diferentes reacciones químicas, pero dada la similitud estructural entre una reacción química y las Redes de Petri se pueden equiparar los elementos de la fórmula y utilizarlos para una Red de Petri. La correspondencia de los elementos de la fórmula con las Redes de Petri se define a continuación:

- τ = valor de tau actual calculado con la Ecuación (4.2).
- ϵ_i = Variable de ajuste cuyo valor es 0.03, dado que con este valor devuelve resultados más precisos que con otros valores diferentes [GP03].
- x_i = Suma de las ratios de las transiciones habilitadas.
- v_{ij} = Matriz de incidencia de la Red de Petri.
- a_j = Vector de ratios de cada transición habilitada multiplicado por su "enabling-degree".

4.3. Comparativa de métodos

En esta sección se muestra a través de diferentes gráficas las mejoras de tiempo de simulación que ofrece el nuevo método de simulación aproximada con respecto al método actual que proporciona PeabraiN. Para ello se han realizado diferentes simulaciones sobre las Redes de Petri que se muestran en las Figuras 4.1 y 4.2 ("cinema.xml" y "SDBS.xml") y en cada simulación es incrementado el número de tokens del marcado inicial multiplicándolo un número de veces determinado. Para cada simulación, se mide el *tiempo de ejecución de simulación* y el *throughput* (rendimiento). El valor *throughput* de una transición muestra el número de veces que esta se dispara a lo largo de la simulación dividido por el tiempo total de simulación.

El servidor que se ha utilizado para realizar las simulaciones está virtualizado con VMware y cuenta con 1GB de RAM y 1 procesador Intel Core i5 de 3,1 GHz. El número de iteraciones realizadas para cada red con diferente marcado ha sido de 200, excepto para la red "SDBS.xml" con multiplicador 40, cuyo número de iteraciones ha sido de 50 debido a que el tiempo de ejecución de cada una de ellas supera las 8 horas, y simular 200 iteraciones suponía más de 2 meses de simulación.

La Red de Petri "cinema.xml" [RJM12] representa un sistema de venta de una máquina de palomitas de maíz en un cine. La red "SDBS.xml" [RJM13] representa un Sistema

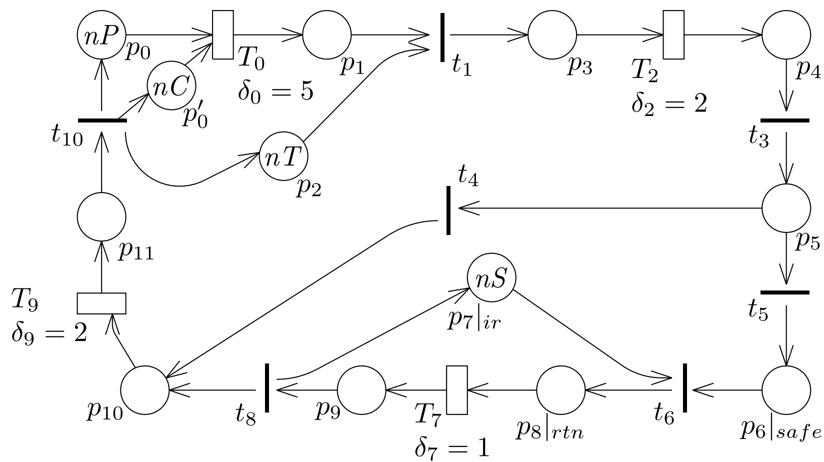

Figura 4.1: Red de Petri “cinema.xml”.

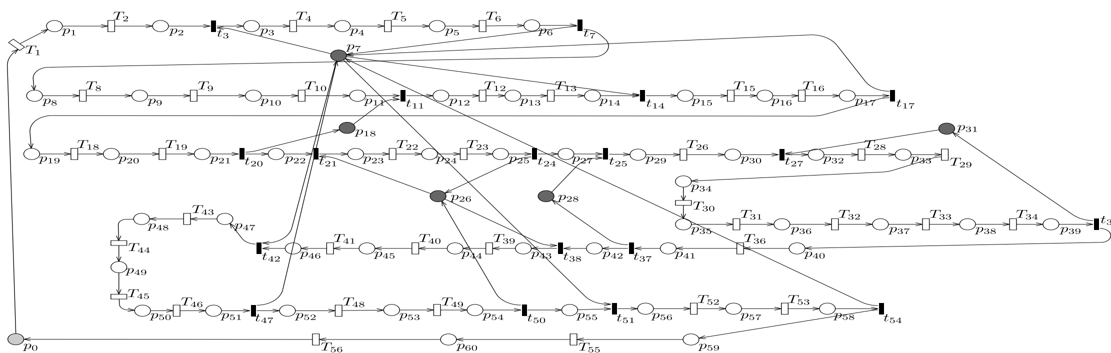

Figura 4.2: Red de Petri “SDBS.xml”.

	Throughput			**Tiempo de simulación**		
Mult.	**Exacto**	**Tau-Leaping**	**%error**	**Exacto**	**Tau-Leaping**	**%mejora**
x1	0,4451	0,5345	20,085 %	1,1547	1,0582	8,35 %
x10	4,4490	4,4529	0,087 %	9,5523	5,3147	44,36 %
x20	8,8992	8,8976	0,018 %	18,8627	10,0144	46,90 %
x30	13,3483	13,3485	0,001 %	27,6070	11,1995	59,43 %
x40	17,7953	17,7973	0,011 %	36,8927	11,1486	69,78 %
x50	22,2523	22,2447	0,034 %	47,2121	11,2684	76,13 %
x75	33,3687	33,3700	0,003 %	68,7478	11,2139	83,68 %
x100	44,4882	44,4949	0,015 %	91,2042	11,1927	87,72 %

Tabla 4.2: Resultados numéricos de la Red de Petri *cinema.xml*.

	Throughput			**Tiempo de simulación**		
Mult.	**Exacto**	**Tau-Leaping**	**%error**	**Exacto**	**Tau-Leaping**	**%mejora**
x1	0,6862	0,9101	32,62 %	390	220	43,58 %
x2	1,4190	1,7198	21,19 %	859	222	74,15 %
x5	3,6367	4,0662	11,81 %	2266	217	90,42 %
x10	7,5610	7,8960	4,4306	4779	211	95,58 %
x20	14,8693	15,4916	4,1851	9478	210	97,78 %
x40	29,7323	30,2366	1,6961	18827	214	98,86 %

Tabla 4.3: Resultados numéricos de la Red de Petri *SDBS.xml*.

de base de datos seguro (Secure Database System). Ambas redes tienen un único T-semiflujo, es decir, todas sus transiciones tienen el mismo valor de *throughput*.

En las Tablas 4.2 y 4.3 se muestran los valores numéricos para todas las simulaciones de ambas redes, así como el porcentaje de mejora de tiempo y el porcentaje de error del *throughput* con respecto al método exacto. En el caso de la red “cinema.xml” se han realizado incrementos multiplicando el marcado por 10, 20, 30, 40, 50, 75 y 100. Dado que la red “SDBS.xml” es bastante mayor que la anterior, los incrementos para esta han sido de 2, 5, 10, 20 y 40. Se puede ver claramente que el método Tau-Leaping supera al método actual cuando se observan los tiempos de ejecución (independientemente del marcado). Además, esta mejora del tiempo de ejecución es claramente dependiente de la complejidad de la red, siendo cada vez más sustancial en cuanto la red es más compleja (o tiene mayor número de marcas).

Para visualizar la magnitud de la mejora es preciso mostrar los resultados de forma gráfica. En la Figura 4.3 se observa de forma muy clara la mejora de tiempo de ejecución utilizando el nuevo método *Tau-Leaping*. Para ambas redes el tiempo de ejecución con el método exacto (línea azul) se incrementa de forma lineal conforme se va incrementando el número de tokens sobre el marcado inicial. Por el contrario, el incremento de tiempo aplicando el método Tau-Leaping llega a un valor máximo que deja de crecer independientemente de que se aumente el marcado de la Red de Petri, mostrando una línea horizontal verde en ambas gráficas.

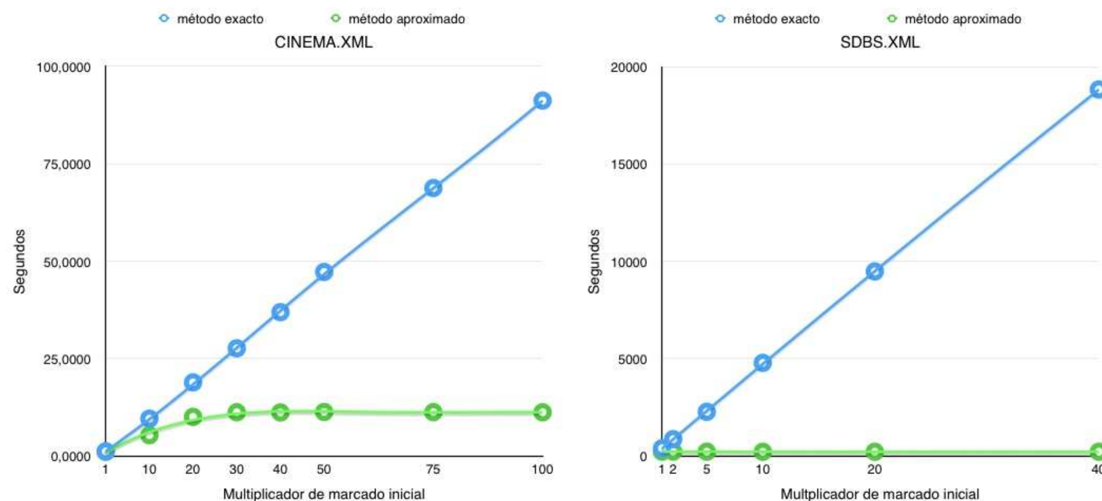

Figura 4.3: Comparativa de tiempos de ejecución de ambas redes utilizando los dos métodos.

La gráfica del *throughput* se muestra en la Figura 4.4. En ambas redes sólo se puede observar una única línea, aunque realmente son dos rectas superpuestas casi idénticas. Esto es debido a que los valores obtenidos por cada método son prácticamente iguales, no obstante, en las simulaciones realizadas sobre la red más compleja “SDBS.xml” si que se observa que la línea verde del método aproximado no coincide exactamente con la línea azul del método exacto, pero de igual modo son unos resultados muy parecidos. Esta pequeña diferencia es menos significativa conforme se incrementa el marcado inicial. El error relativo de *throughput* entre ambos métodos en las dos redes se muestra en la Figura 4.5. Como se observa, el error de *throughput* no varía apenas conforme se incrementa de marcado inicial, e incluso para la red “cinema.xml” dicho error se mantiene muy próximo a cero a pesar de los diferentes incrementos de marcado.

Como conclusión de los resultados, se observa que el método de Tau-Leaping es mejor para redes grandes con mucha población (número de marcas) que el método de simulación exacta actual.

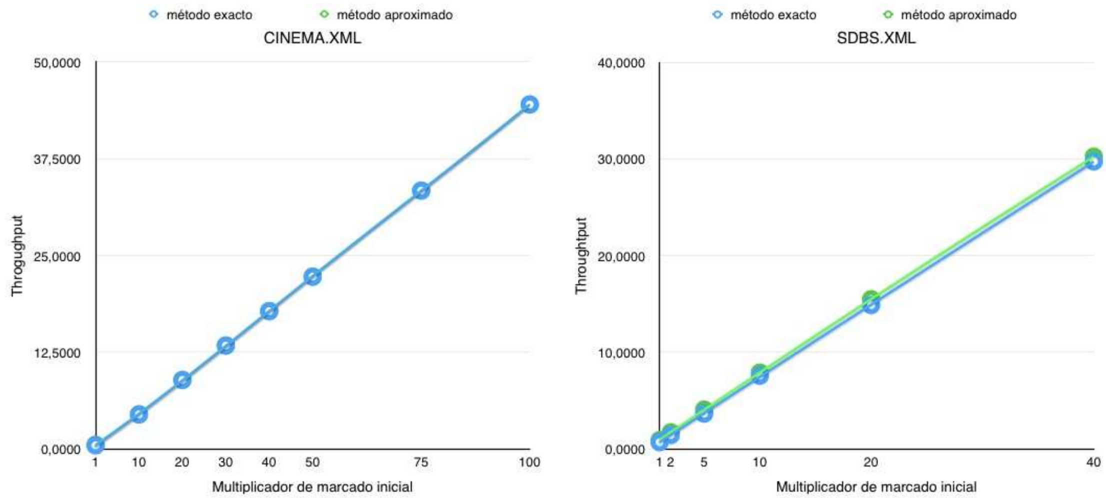

Figura 4.4: Comparativa de *throughput* (rendimiento) de ambas redes utilizando los dos métodos.

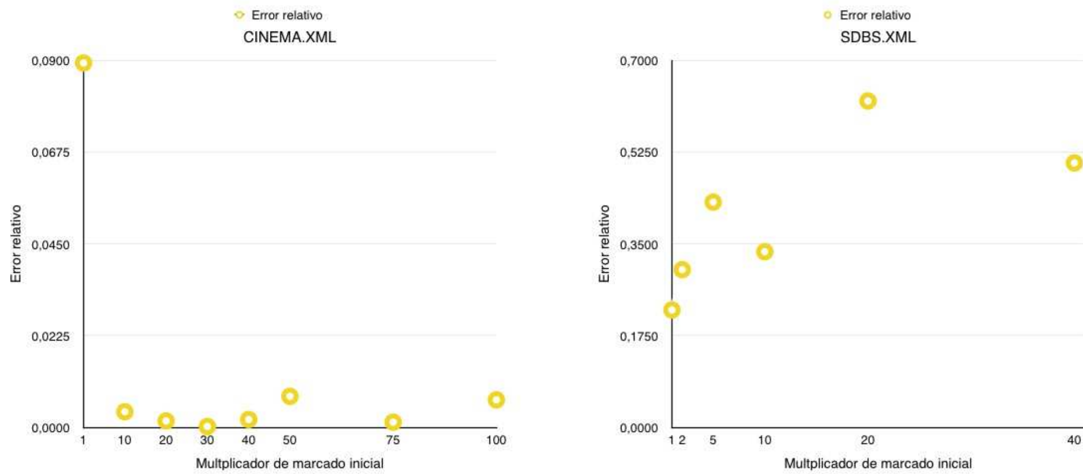

Figura 4.5: Error relativo de *throughput* para RdP “cinema.xml” y “SDBS.xml”.

Capítulo 5

Trabajo Relacionado

Este capítulo reúne los trabajos previos que tienen relación con la simulación aproximada de Redes de Petri.

Con respecto a las tres primeras fases del proyecto no existen trabajos relacionados, ya que PeabraiN es una herramienta muy específica con poco tiempo de vida y que se encuentra en su primera versión, aunque si existen otras herramientas que también realizan simulaciones estocásticas, como pueden ser Matlab y GreatSPN. Para Matlab existe un módulo desarrollado por "nvictus" (`https://github.com/nvictus/Gillespie`) que realiza simulaciones utilizando los dos métodos formulados por Gillespie, método de la primera reacción y método directo. GreatSPN permite crear y editar Redes de Petri para realizar simulaciones exactas, pero en algunas pruebas realizadas se ha observado que los tiempos de simulación obtenidos con GreatSPN son mucho mayores que los tiempos conseguidos por cualquiera de los dos métodos que se han utilizado en este proyecto. Además, GreatSPN también permite calcular cotas de rendimiento superiores e inferiores de las transiciones al igual que hace PeabraiN a través del módulo *Lower (Upper) throughput bound*.

Existen trabajos que usan el método de simulación escolástico de Tau-Leaping con Redes de Petri, como por ejemplo [CMMA13], donde se simula el flujo de coagulación modelado con Redes de Petri. Sin embargo, el simulador que se usa en [CMMA13] no está disponible de forma libre, a diferencia del que se ha desarrollado en este trabajo.

Capítulo 6

Conclusiones

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este proyecto así como posibles trabajos futuros con los que mejorarlo.

PeabraiN vio la luz hace muy poco tiempo (2012) y por lo tanto tiene todavía mucho recorrido por delante. El potencial de la herramienta es grande y en este proyecto se ha dado un primer paso para mejorarla. Las mejoras aparentemente tienen poca relación entre sí: Cada una de ellas esta enfocada en *mejorar la herramienta* como tal (selector de solucionadores), otra de ellas *mejora la interactividad con el usuario* aportando una nueva manera de introducir los cálculos (CLI), y finalmente la última mejora *pone a la herramienta al alcance de todo el mundo* a través de Internet. Sin embargo, cada mejora está complementada por la anterior: En el caso del CLI (Fase 2), utiliza el selector de solucionadores desarrollado en la primera fase, y en el caso de PeabraiN Web (Fase 3), utiliza por debajo el CLI desarrollado en la Fase 2 para lanzar los cálculos. La reutilización de funcionalidades ha sido algo que ha facilitado y reducido enormemente el trabajo llevado a cabo en el PFC.

En la mejora llevada a cabo en la parte de simulación estocástica se han mejorado notablemente los tiempos de simulación con respecto al método anterior y esto puede dar lugar a realizar simulaciones con Redes de Petri de gran tamaño que hasta ahora no era posible realizar en un período de tiempo razonable.

Como resultado global del proyecto he ampliado de manera considerable muchos conocimientos que a lo largo de la carrera no se dan con total profundidad, entre algunos de ellos, he conseguido amplios conocimientos de Java (uno de los lenguajes más extendidos actualmente) además de familiarizarme ampliamente con el entorno de desarrollo *Eclipse*, he asentado y aumentado mi entendimiento sobre el estándar UML tan utilizado en cualquier proyecto de desarrollo, he profundizado en las Redes de Petri y he aprendido mucho acerca de las simulaciones estocásticas.

Otro concepto adquirido a raíz de este proyecto es que un buen diseño, y una buena organización y planificación son elementos a tener muy en cuenta a la hora de realizar cualquier proyecto, ya que facilitan enormemente el trabajo y ahorran gran cantidad de tiempo y esfuerzo.

Los trabajos futuros que se pueden realizar para complementar o mejorar este pro-

yecto se pueden dividir en función de los grandes ámbitos de este proyecto. En la parte de PeabraiN se pueden seguir desarrollando nuevos módulos que amplíen las funcionalidades de la herramienta y sobre PeabraiN WEB se pueden mejorar sus funcionalidades para poder programar diferentes cálculos sobre diferentes redes de una sola vez, y que los resultados finales sean enviados a un usuario (previamente registrado) por correo electrónico. En la parte de simulación estocástica se pueden implementar nuevos métodos y realizar una comparativa más completa entre todos los métodos de simulación que existen actualmente tomando una muestra mayor de Redes de Petri.

Bibliografía

[AMBC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, 1995.

[Cam98] Javier Campos. Performance Bounds. In G. Balbo and M. Silva, editors, *Performance Models for Discrete Event Systems with Synchronizations: Formalisms and Analysis Techniques, Chapter 17*, chapter 17, pages 587–635. Editorial KRONOS, Zaragoza, Spain, September 1998.

[CMMA13] Davide Castaldi, Daniele Maccagnola, Daniela Mari, and Francesco Archetti. Stochastic Simulation of the Coagulation Cascade: A Petri Net Based Approach. In Ioannis Caragiannis, Michael Alexander, RosaMaria Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, StephenL Scott, and Josef Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 248–262. Springer Berlin Heidelberg, 2013.

[CS92] J. Campos and M. Silva. Structural Techniques and Performance Bounds of Stochastic Petri Net Models. *Lect. Notes Comput. Sc.*, 609:352–391, 1992.

[DKS] Nicholas J. Dingle, William J. Knottenbelt, and Tamas Suto. Pipe2: A tool for the performance evaluation of generalised stochastic petri nets.

[Gil76] Daniel T Gillespie. A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions. *J. Comput. Phys.*, 22(4):403–434, 1976.

[Gil07] Daniel T. Gillespie. Stochastic simulation of chemical kinetics. *Annual Review of Physical Chemistry*, 58(1):35–55, 2007.

[Gil08] Daniel T. Gillespie. Simulation methods in systems biology. In Marco Bernardo, Pierpaolo Degano, and Gianluigi Zavattaro, editors, *SFM*, volume 5016, of *Lecture Notes in Computer Science*, pages 125–167. Springer, 2008.

[GIS] GISED. Group of Discrete Event Systems Engineering. [Online]. `http://webdiis.unizar.es/GISED/`.

[glp06] GLPK (GNU linear programming kit), 2006.

[GP03] D. T. Gillespie and L. R. Petzold. Improved leap-size selection for accelerated stochastic simulation. *J. Chem. Phys.*, 119:8229–8234, October 2003.

[Gro11] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.4.1. Technical report, August 2011.

[Hol08] Anthony T. Holdener, III. *Ajax: The Definitive Guide.* O'Reilly, first edition, 2008.

[Mur89] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–576, 1989.

[OMG07] OMG. *XML Metadata Interchange (XMI).* OMG, 2007.

[Pip] Pipe2. PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. [Online]. `http://pipe2.sourceforge.net`.

[RJM12] Ricardo J. Rodríguez, Jorge Júlvez, and José Merseguer. PeabraiN: A PIPE Extension for Performance Estimation and Resource Optimisation. In *Proceedings of the 12th International Conference on Application of Concurrency to System Designs (ACSD)*, pages 142–147. IEEE, 2012.

[RJM13] Ricardo J. Rodríguez, Jorge Júlvez, and José Merseguer. On the Performance Estimation and Resource Optimisation in Process Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(6):1385–1398, 2013.

Apéndice A

Fases de Desarrollo

A.1. Diagrama de *Gantt*

En este capítulo se proporciona la información sobre la distribución temporal de cada una de las fases de este proyecto, así como el tiempo invertido en cada una de ellas en las que fue dividido. El tiempo total dedicado al proyecto ha sido de 460 horas y se puede ver desglosado en la tabla.

Tareas	**Horas**
Reuniones	30
Búsqueda de información	50
Fase 1 - Selector de solucionadores	75
Fase 2 - CLI	85
Fase 3 - PeabraiN Web	35
Fase 4 - Simulación Estocástica	75
Documentación memoria	110
Total	**460**

Tabla A.1: Horas dedicadas.

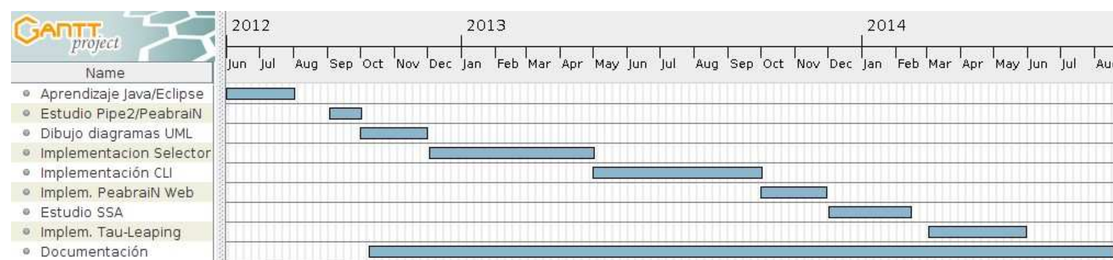

Figura A.1: Diagrama de Gantt del desarrollo del proyecto.

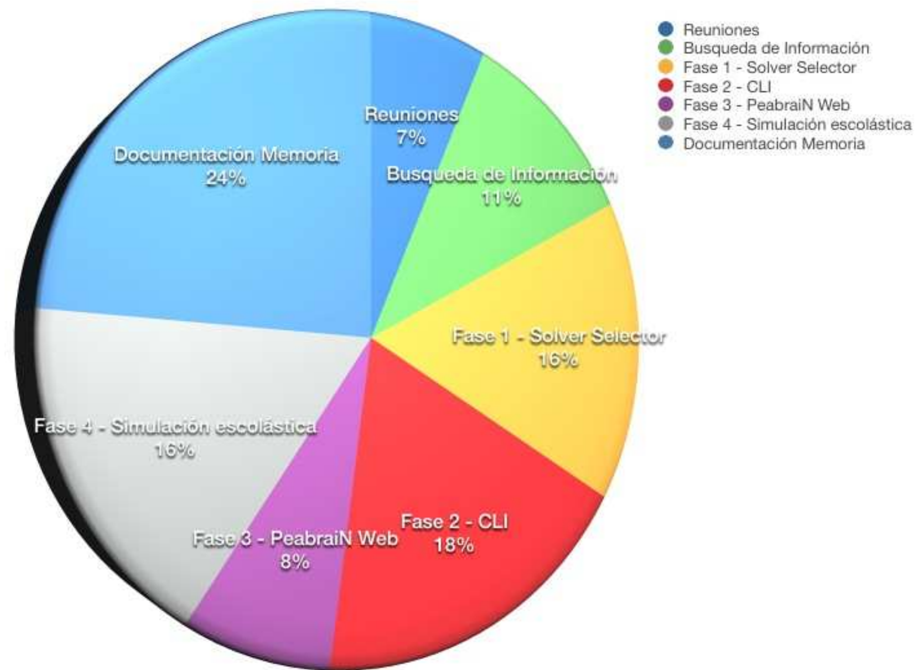

Figura A.2: Distribución del esfuerzo en las diferentes fases del proyecto.

El proyecto ha sido realizado entre Abril de 2012 y Agosto de 2014. El total de horas invertidas en el proyecto asciende a 460 horas.

Apéndice B

Código fuente del simulador estocástico aproximado (método Tau-Leaping)

```
package pipe.modules.gspnsimulator;

import java.util.ArrayList;
import java.util.Random;

import pipe.modules.bounds.dataLayer.PetriNetModel;
import umontreal.iro.lecuyer.probdist.PoissonDist;

public class ApproximateSimulator extends Simulator {

  public ApproximateSimulator(PetriNetModel pnModel) {
    super(pnModel);
  }

  @Override
  protected double runSimulationMethod(int[] marking, double maxSimulation,
      double[] obsTrans) {
    double simTime = 0.0;
    double tau = 0.0;

    TransitionEventList aux = null;

    marking = fireImmediateTransitionsAt(marking, simTime);

    ArrayList<Integer> enabledTransitions = getEnabledTransitionsAt(TransitionType.
        TIMED_TRANS, marking);
    tau = calculateTau(marking, enabledTransitions);
    aux = generateEventList(marking, enabledTransitions, tau);

    if (aux.al.size() == 0) {
      System.err.println("Check initial marking. No evolving is possible.");
```

```
        return 0;
    }

    // NextEvent ne = getNextEvent(aux);
    for (int i = 0; i < aux.al.size(); i++) {
        marking = firesTransitionAt(aux.al.get(i).idxTransition, marking, (int) aux.al.get(i).minValue,
             simTime);
    }

    do {
        simTime += tau;
        marking = fireImmediateTransitionsAt(marking, simTime);
        enabledTransitions = getEnabledTransitionsAt(TransitionType.TIMED_TRANS, marking);
        tau = calculateTau(marking, enabledTransitions);
        aux = generateEventList(marking, enabledTransitions, tau);

        for (int i = 0; i < aux.al.size(); i++) {
            marking = firesTransitionAt(aux.al.get(i).idxTransition, marking, (int) aux.al.get(i).minValue
                 , simTime);
        }

    } while (simTime <= maxSimulation);

    // End of simulation...
    return simTime;
}

private TransitionEventList generateEventList(int[] marking, ArrayList<Integer>
     enabledTransitions, double tau) {

    Random r = new Random();
    //double tau = CalculateTau(marking,enabledTransitions);
    ArrayList<TransitionEvent> v = new ArrayList<TransitionEvent>(enabledTransitions.size());
    int idx, factor;
    double random, lambda, aux = 0;

    for (int i = 0; i < enabledTransitions.size(); i++) {
        idx = (int) enabledTransitions.get(i);
        factor = 1;
        // Change factor of exp, if needed (depends on the semantics!) for sum of ratios
        if (pnModel.isInfiniteSemantics(idx))
            factor = pnModel.getEnablingDegree(marking, idx);

        // Tau-Leaping computation
        random = r.nextDouble();
        lambda = pnModel.getTransitionRate(idx) *
             pnModel.getEnablingDegree(marking, idx) * tau;

        // Get sum of ratios
        aux += factor * pnModel.getTransitionRate(idx);
        v.add(new TransitionEvent(idx, PoissonDist.inverseF(lambda, random), 0));
    }
```

```
    return new TransitionEventList(v, aux);
  }

  private double calculateTau (int[] marking, ArrayList<Integer> enabledTransitions){
    double a0 = 0; // Sum of Ratios
    double epsilon = 0.03;
    int idx, factor;

    // get a0 (sum of ratios)
    for (int i = 0; i < enabledTransitions.size(); i++) {
      idx = (int) enabledTransitions.get(i);
      factor = 1;
      // Change factor of exp, if needed (depends on the semantics!)
      if (pnModel.isInfiniteSemantics(idx))
        factor = pnModel.getEnablingDegree(marking, idx);

      a0 += factor * pnModel.getTransitionRate(idx);
    }

    int incidenceMatrix[][] = this.pnModel.getIncidenceMatrix(),
      auxIdxTrans, auxIncidence, auxIdxPlace;

    double  tau = Double.MAX_VALUE,
        tautemp = Double.MAX_VALUE;

    // for all enabled transitions
    for (int i = 0; i < enabledTransitions.size(); i++) {
      ArrayList<Integer> places = this.pnModel.findInColPre(enabledTransitions.get(i));

      // for all places pre transition
      for (int ii = 0; ii < places.size(); ii++) {
        auxIdxPlace = places.get(ii);
        ArrayList<Integer> transitionPRE = this.pnModel.findInRowPost(auxIdxPlace); // todas
            las transiciones previas al lugar
        ArrayList<Integer> transitionPOST = this.pnModel.findInRowPre(auxIdxPlace); // todas
            las transiciones posterior al lugar
        double  mu = 0, auxRate, auxEnabDegree,
            sigma = 0;
        for (int iii = 0; iii < transitionPRE.size(); iii++) {
          auxIdxTrans = transitionPRE.get(iii);
          if (!this.pnModel.isImmediate(auxIdxTrans))
          {
            auxRate = this.pnModel.getTransitionRate(auxIdxTrans);
            auxEnabDegree = this.pnModel.getEnablingDegree(marking, auxIdxTrans);
            auxIncidence = incidenceMatrix[auxIdxPlace][auxIdxTrans];
            mu = mu + (auxRate* auxEnabDegree*auxIncidence);
            sigma += auxRate* auxEnabDegree*(auxIncidence*auxIncidence);
          }
        }
        for (int iii = 0; iii < transitionPOST.size(); iii++) {
          auxIdxTrans = transitionPOST.get(iii);
          if (!this.pnModel.isImmediate(auxIdxTrans))
          {
```

```
                auxRate = this.pnModel.getTransitionRate(auxIdxTrans);
                auxEnabDegree = this.pnModel.getEnablingDegree(marking, auxIdxTrans);
                auxIncidence = incidenceMatrix[auxIdxPlace][auxIdxTrans];
                mu = mu + (auxRate* auxEnabDegree*auxIncidence);
                sigma += auxRate* auxEnabDegree*(auxIncidence*auxIncidence);
            }
        }
        // Get minimum (see "Simulation Methods in System Biology", D. T. Gillespie, FMCSB
            2008, page 17)
        double num1 = (Math.max(1, epsilon * a0)) / Math.abs(mu),
            num2 = (Math.pow(Math.max(1, epsilon * a0), 2)) / sigma,
            num = Math.min(num2, num1);

        // Update tautemp
        if (num < tautemp) {
            tautemp = num;
        }
    }
    // Get minimum tau
    if (tau > tautemp)
        tau = tautemp;
  }

  return tau;
}

@Override
protected SimulationMethod getMethod() {
  return SimulationMethod.APPROX_SIM;
}
}
```